Simple concurrent garbage collection almost without synchronization

Hesselink, Wim H.; Lali, M.I.

Published in:
Formal methods in system design

DOI:
10.1007/s10703-009-0083-z

Publication date:
2010

Form Methods Syst Des (2010) 36: 148–166
DOI 10.1007/s10703-009-0083-z

Simple concurrent garbage collection almost without synchronization

Wim H. Hesselink · M.I. Lali

Published online: 12 September 2009

Abstract We present two simple mark and sweep algorithms, A and B, for concurrent garbage collection by a single collector running concurrently with a number of mutators that concurrently modify shared data. Both algorithms are based on the ideas of Ben-Ari's classical algorithm for on-the-fly garbage collection with one mutator. The algorithms require the mutators to estimate the set of objects they currently hold in private variables. They differ in the grain of atomicity of this estimate and in their mutator marking requirements.

For algorithm A, the only synchronization needed is at the point where the list of newly collected garbage nodes is included in the free list to again become available to the mutators. Such synchronization of access to the free list seems unavoidable. For algorithm A, the estimate can be constructed concurrently with mutator actions. Algorithm B requires that the estimate is constructed atomically.

Algorithms of this form have always been error-prone. We therefore provide assertional proofs of correctness, which moreover have been verified with the proof assistant PVS.

Keywords Concurrent garbage collection · Concurrent algorithms · Verification · Theorem proving · Memory model

1 Introduction

We consider the following situation. A number of processes (threads) is concurrently active in shared memory, allocating new objects and modifying object fields, i.e., pointers to other objects. Allocated objects that can no longer be reached by processes are called *garbage*. The aim of garbage collection is to reclaim the memory these garbage objects

W.H. Hesselink (✉) · M.I. Lali
Dept. of Mathematics and Computing Science, University of Groningen, P.O. Box 407,
9700 AK Groningen, The Netherlands
e-mail: w.h.hesselink@rug.nl
url: http://www.cs.rug.nl/~wim

M.I. Lali
e-mail: m.i.ullah@student.rug.nl

occupy. We propose two new algorithms where a single garbage collecting process concurrently traverses the memory to gather unreachable objects for reclamation in a so-called mark-and-sweep cycle.

In this context, the processes that allocate new objects and modify object fields are called *mutators* while the garbage collecting process is called the *collector*. The memory is thus concurrently inspected and modified by several mutators and one collector. In order to program the collector correctly, we need assumptions on the actions of the mutators as well as assumptions on whether the collector can inspect private memory of the mutators. The stronger we limit the activities of the mutators and the more powerful inspection rights we grant the collector, the easier it is to program the collector. In [8], we allowed several concurrent collectors, and limited their inspection rights and the mutators' actions considerably. Here, we consider a single collector that can ask for an estimate of the nodes held by any mutator, but the memory modifications of the mutators are unrestricted.

How do mutators inspect and modify the memory? By reading shared data into private variables, by allocating objects for private variables, and by modifying fields of objects that are referenced to by private variables. The fundamental paper [6] introduced a free list such that allocating new objects is nothing but pointer modification. This was an important simplification, taken over by many of the subsequent papers, e.g. [5, 11].

The key-idea of Ben-Ari's algorithm [5] is of counting the number of objects that become black during a call of the marking procedure *mark*. When this number is zero, the call is called *nonblackening*. In [5], one nonblackening call is enough to ensure that all reachable objects are black. In the concurrent setting of [11], M or $M+1$ nonblackening calls of *mark* are needed for this, where M is the number of mutators. Our algorithms again need only one nonblackening call.

The classical algorithms [5, 6, 11] assume that the mutators only access objects reachable from global roots. As pointed out by [7, 3.2], this assumption is unrealistic and undesirable. Following [7], we therefore take the following starting point.

The mutators access the shared data via *local references*. For any mutator m, we define the set $m.Lref$ to consist of the objects mutator m holds references to. For algorithm A, we postulate a procedure $getLref(m)$ that gives a well-defined estimate of $m.Lref$, see specification (0) in Sect. 2.1. Mutator m need not be suspended during $getLref(m)$. We leave the implementation of *getLref* to the implementor. For algorithm B, $getLref(m)$ has to construct an atomic snapshot of $m.Lref$, just as in [7, 13].

Although we speak of a single collector, the collector code contains several large loops that can easily be parallelized because they are instances of the write-all problem (e.g. [9]). If one wants to exploit this possibility, the collaborating collectors would need a barrier (e.g. [1]) to synchronize. In the remainder of this paper, we keep to a single collector.

Related work The classical case [6, 19] is that of a single garbage collector concurrently acting with a single mutator. The paper [19] uses semaphores and several extended PV sections, whereas [6] eliminates the need for additional synchronization. Following [5, 6, 20], we call this *on-the-fly garbage collection*. Confusingly, it is called *real-time* garbage collection in [22] and *concurrent* garbage collection in [21].

We speak of *concurrent garbage collection* when there is a single collector concurrently acting with several concurrent mutators. The present algorithms belong to this class, as do those of, e.g., [4, 7, 11, 13].

In particular, our algorithm B can be compared with the algorithm of Doligez and Gonthier [7]. This DG-algorithm uses four colours instead of two. The mutators share more variables with the collector than in our algorithm. The DG-mutator action that corresponds

to our *mutate* requires several accesses of shared variables, much more than the single blackening action of *mutate* in our version B. In conclusion, our algorithm B is simpler and less optimized, and requires less cooperation of the mutators. Our collector has more work to do, but can be parallelized more easily.

There are several recent studies on reference counting concurrent collectors, e.g., [4, 13]. A disadvantage of reference counting collectors is that they do not collect all cycles. In [13, p. 5, 6], the remedy is to concurrently but seldom apply the concurrent mark-and-sweep algorithm of [3]. Another proposed remedy is the application of a special cycle collecting algorithm [14, 18].

Several (concurrent) garbage collection algorithms have been verified mechanically. For example, McCreight a.o. [15] provide a general framework for certifying garbage collectors by means of the theorem prover Coq. The algorithms of [5, 11] have been verified with Isabelle/HOL in [16]. We used the proof assistant PVS [17] throughout the design, see [10].

Overview In Sect. 2, we present the garbage collecting algorithms A and B informally. The algorithms are formalized and proved in Sect. 3. In Sect. 4 we briefly describe our verification by means of the proof assistant PVS. Conclusions are drawn in Sect. 5.

2 Informal description of both algorithms

In Sect. 2.1, we describe the setting and the assumptions for both algorithms.

In Sect. 2.2, we describe two procedures to mark the reachable objects. In the absence of interleaving mutator actions, procedure *markApp* efficiently marks all reachable objects. In the presence of mutator actions, it still efficiently marks reachable objects, but there is no guarantee that it marks all of them. Procedure *mark* is less efficient, but can be used to verify that all reachable objects are marked.

Indeed, in Sect. 2.3, we present two ways to adapt the mutator actions in such a way that a call of procedure *mark* guarantees that all reachable objects have been marked if the call does not newly mark any unmarked objects. The correctness of this adaptation is informally discussed in Sect. 2.4.

In Sect. 2.5, we describe the remainder of the marking phase, which is analogous to Ben-Ari's classical algorithm [6]. In Sect. 2.6, we roughly describe the free list. Here, we are somewhat more explicit than the classical treatments [5, 6, 11].

2.1 Setting

We assume that a finite set *Obj* of shared objects is given. Every object x has a fixed set $x.F$ of fields, the values of which can be modified. We define *Edge* to be the set of pairs (x, f) with $x \in Obj$ and $f \in x.F$. We assume that there are two special objects `free` and $\perp$, and that $\texttt{free}.F \neq \emptyset$ and $\perp.F = \emptyset$. The object `free` serves as a header for a number of free lists, $\perp$ serves as `null`.

The state of the memory is characterized by a mutable function $h : Edge \to Obj$. Function h indicates the values of all fields: $h(x, f)$ is the current value of field f of object x. As is usual, e.g. [21], we often abbreviate $h(x, f)$ as $x.f$. There is a finite set *Mut* of mutator processes that perform modifications of the data structure.

Definition For each mutator m, we define $m.Lref$ as the set of objects, mutator m has *local references* to, i.e., the set of pointer valued fields in the stack frames currently active. Mutator m can access and modify the shared data only via $m.Lref$. The collector can ask any mutator m for the current value of the set $m.Lref$.

In the case of algorithm A, we assume that a procedure $getLref(m)$ is given, which can be called by *collector*, executes concurrently with all mutators, and returns a set of objects N that estimates $m.Lref$ without the need for synchronization. We use the following specification:

(0) $getLref(m : Mut)$ **returns** $N : \mathbb{P}(Obj)$
$\{m.Lref^- \subseteq N \subseteq m.Lref^+\}$,

where $m.Lref^-$ is the set of objects x such that, at the call of $getLref(m)$, mutator m has a local reference to x and that during the execution no local reference to x is deleted or re-assigned; $m.Lref^+$ is the set of objects x such that m has a local reference to x at some time during the call. We model this by treating $m.Lref^-$ and $m.Lref^+$ as private variables of mutator m, which are reset to $m.Lref$ at the call of $getLref(m)$. Note that $m.Lref^-$ and $m.Lref^+$ need not occur in the implementation, they only serve in the model to specify $getLref$.

For algorithm B, we specify that $getLref(m)$ yields a copy of $m.Lref$, atomically.

Mutator m can modify its set $m.Lref$ of local references only by inspection of the data structure from a given local reference, and by deleting or re-assigning local references. These actions also affect the values of $m.Lref^-$ and $m.Lref^+$, in the following way.

$m.load(x : m.Lref,\ f : x.F)$:
$\quad \langle add\ x.f\ to\ m.Lref\ and\ m.Lref^+ \rangle$,
$m.delete(x : m.Lref)$:
$\quad \langle remove\ x\ from\ m.Lref^-\ and\ possibly\ from\ m.Lref \rangle$.

The brackets $\langle\ \rangle$ are atomicity brackets to indicate that the command enclosed by them is to be executed atomically. Note that *delete* removes x from $m.Lref^-$, but that x can remain in $m.Lref$ when some local reference to x is deleted or re-assigned while other local references to x remain. We model *delete* nondeterministically to abstract from the number of local references to x. We could model *Lref* as a multiset [7], but that complicates matters without gaining us anything.

The sets $m.Lref$ are often unions of stack frames. Therefore, rather than deleting single elements, the mutators may delete sets of references in a single step. Yet, we can model set deletions as sequential compositions of single deletions.

For the sake of allocation and garbage collection, we assume that there are two special objects `free` and $\perp$, that are never collected as garbage. Indeed, `free` must remain available as a header for the garbage to be collected, and $\perp$ at least as a tail for the free list. In applications, it may be convenient to postulate some other *global roots* [13, 7.1] that are never collected as garbage, e.g., for the sake of data communication. We therefore assume that a fixed set *Common* of objects is given, that satisfies:

$$\texttt{free}, \perp \in Common.$$

We define the set *Sources* of the *immediately reachable* objects by

$$Sources = \{x \mid x \in Common \ \vee\ (\exists\, m \in Mut : x \in m.Lref)\}.$$

We associate to function h the directed graph of the objects with edges given by the values of the fields, i.e., the binary relation $[h]$ on Obj given by

$$[h] = \{(x, y) \mid \exists\ f : h(x, f) = y\}.$$

Let $[h]^*$ be the reflexive transitive closure of $[h]$. We define the set *Reach* of the *reachable* objects by

$$Reach = \{y \mid \exists\, x \in Sources : (x, y) \in [h]^*\}.$$

Mutator m can modify the shared data, i.e., function h, by

$$m.mutate(x, y : m.Lref, f : x.F)\colon \langle x.f := y\rangle.$$

Mutator actions never let the set *Reach* grow. *Reach* can shrink by the actions *delete* and *mutate*. The aim of garbage collection is to reclaim all unreachable objects, but no reachable ones.

Example 1 As an illustration of how ordinary mutator actions are encoded in the ones provided above, we consider a mutator that reverses a *next*-connected list that ends in $\perp$ by means of the procedure

```
reverse(var x) =
  if x ≠ ⊥ then
    var y := x.next    (load);
    x.next := ⊥    (mutate);
    while y ≠ ⊥ do
      var z := y.next    (load);
      y.next := x    (mutate);
      x := y ; y := z    (delete)
  end end (delete).
```

In the body of the while loop, the assignments to z and $y.next$ are calls of *load* and *mutate*, respectively, and the assignments to x and y are implicit deletions from *Lref*. The final deletion corresponds to the deletion of the stack frame at the return of the call.

When the nonatomic version of *getLref* is called and returns during a call of *reverse*, it may return $\{\perp\}$ because the mutator does not own any other object continuously during the call. In particular, the resulting set N need not contain the head of the reversed list. Therefore, the collector cannot rely on the nonatomic version of *getLref* to completely determine the set *Sources*.

2.2 Marking

Every garbage collecting algorithm that works by mark and sweep consists of a marking phase to mark all reachable objects and a *sweep* that reclaims the unmarked objects and adds them to the free list.

```
collector() =
  while true do markingPhase(); sweep end.
```

We therefore assume that every object x has a boolean field b (for *black*) to mark it. We define the state function *Black* as an alias of the set $\{x \in Obj \mid x.b\}$. The marking phase needs to satisfy the specification:

(1) *markingPhase*();
$\{Reach \subseteq Black \subseteq Reach^0\}$,

where $Reach^0$ is the values of *Reach* at the start of the marking phase. We shall use the second inclusion to show the absence of memory leaks in the sense that all garbage is eventually collected.

The definition of the set *Reach* implies that it is the smallest set X of objects that satisfies

$$\begin{array}{l} \mathit{Sources} \subseteq X, \\ \forall\,(x, f) \in \mathit{Edge} : x \in X \quad \Rightarrow \quad x.f \in X. \end{array}$$

We therefore try to establish the predicates

$$\begin{array}{ll} \mathit{Lq0}: & \mathit{Sources} \subseteq \mathit{Black}, \\ \mathit{Lq1}: & \forall\,(x, f) \in \mathit{Edge} : x.b \quad \Rightarrow \quad x.f.b. \end{array}$$

These predicates clearly imply $\mathit{Reach} \subseteq \mathit{Black}$.

It is not difficult to make an approximating marker that satisfies the specification

$$\begin{array}{ll} (2) & \mathit{markApp}() \\ & \{\mathit{Common} \subseteq \mathit{Black} \subseteq \mathit{Reach}^0\}, \end{array}$$

where Reach^0 is the values of *Reach before* the call of *markApp*, and that establishes $\mathit{Black} = \mathit{Reach}$ when interleaving mutator actions are absent. An example is:

```
markApp() =
    for all u ∈ Obj do ⟨u.b := false⟩ end;
    obs := Common;
    for all p ∈ Mut do obs := obs ∪ getLref(p) end;
    while obs ≠ ∅ do
        extract some u from obs;
        if ⟨¬u.b⟩ then
            ⟨u.b := true⟩;
            for all g ∈ u.F do ⟨add u.g to obs⟩ end
    end end.
```

The following example shows, however, that when there are interleaving mutator actions, there is no guarantee that *markApp* establishes the more vital inclusion $\mathit{Reach} \subseteq \mathit{Black}$.

Example 2 We consider the following scenario. Let x and y be two objects with $x.f = y$. Assume that mutator m has $m.\mathit{Lref} = \{x\}$. After the second **for** loop, the object y is white and not in *obs*. At that moment, mutator m performs $\mathit{load}(x, f)$. This establishes $m.\mathit{Lref} = \{x, y\}$. Then m performs $\mathit{mutate}(x, x, f)$. We then have $x.f = x$. The collector then executes its **while** loop and blackens x but not y. Therefore *markApp* can terminate with a reachable, but white object y.

We therefore regard *markApp* as an approximating marker that has to be followed by a stricter, but possibly less efficient procedure *mark*. The postcondition of (2) is a precondition for procedure *mark*, and must be preserved as an invariant. The predicates *Lq0* and *Lq1* may suggest the following procedure:

```
        mark() =
E:          for all (u, g) ∈ Edge do
                if ⟨u.b⟩ then ⟨u.g.b := true⟩ end end;
M:          for all p ∈ Mut do
                obs := getLref(p);
                for all u ∈ obs do ⟨u.b := true⟩ end end.
```

Note that we let loop E over the *edges* precede loop M over the *mutators*. This is intentional. At the end of Sect. 2.4, we show that this perhaps unexpected order is better than the reverse one.

Loop E is taken over from [11] because of its simplicity. An equivalent and more efficient version would use a conditionally nested loop:

E': **for all** $u \in Obj$ **do**
if $\langle u.b \rangle$ **then for all** $g \in u.F$ **do** $\langle u.g.b := true \rangle$ **end end**
end.

Since this clutters the proof unnecessarily, we stick to the above version.

One of the things to know about *mark* is that it preserves the predicates *Lq0* and *Lq1* in the following sense:

Theorem 1 *Assume that Lq0 and Lq1 hold at some moment before or during execution of mark. Then they remain valid and the set Black remains constant.*

Proof (Sketch) Execution of *load* preserves *Lq0* because of *Lq1*. Execution of *mutate* preserves *Lq1* because of *Lq0*. At this point, the set *Black* is not changed by the mutators. It is not changed by loop E because of *Lq1*. It is not changed by loop M because *Lq0* implies that the sets *obs* are all contained in *Black*. Because procedure *mark* does not change *Black*, it preserves *Lq0* and *Lq1*. □

Strictly speaking, the formulation of Theorem 1 is too sloppy. We give a stricter version as Theorem 4 in Sect. 3.

2.3 Mutators must blacken

More vital than Theorem 1 is the reverse implication, viz. that, if the set *Black* remains constant during a call of *mark*, the postcondition of this call satisfies *Lq0* and *Lq1*. In the present setting, however, this is not true. We give two example scenarios.

Example 3 Let x and y be two objects with $x.b \wedge \neg y.b$ and $x.f = x$ initially. Loop E preserves this. Assume mutator m has $m.Lref = \{x, y\}$. After loop E, mutator m calls $mutate(x, y, f)$ followed by $delete(y)$. This can establish $x.f = y$ and $m.Lref = \{x\}$. Then loop M is executed, not blackening any white node. The call of *mark* therefore does not blacken any white node, but its postcondition violates *Lq1* because $x.f = y$. Note that this scenario is even applicable when the collector gathers all sets $m.Lref$ in a single atomic step.

Example 4 Let x and y be objects with $x.b$ and $\neg y.b$. Let mutator m have three local references ℓ_1, ℓ_2 and ℓ_3 with the initial values $\ell_1 = \ell_2 = x$ and $\ell_3 = y$. During the call of $getLref(m)$, mutator m performs the assignments $[\ell_1 := \ell_3 \; ; \; \ell_3 := \ell_2]$. According to specification (0), $getLref(m)$ is allowed to deliver $\{x\}$, because the assignment $\ell_3 := \ell_2$ is an implicit deletion of y from $m.Lref^-$. Indeed, this result is obtained when *getLref* just reads the values of ℓ_1, ℓ_2, ℓ_3 in this order and the mutator assignments are done just before *getLref* reads ℓ_3. The call of *mark* therefore need not blacken y even though y remains reachable.

These two examples show that, in order to establish the proposed reverse implication, we need to modify either procedure *mark*, or procedure *delete*, or both procedures *getLref* and *mutate*. We shall retain *mark* and propose two versions: one in which *delete* is modified and one with modifications in both *getLref* and *mutate*. Before doing so, we note that Theorem 1 remains valid when we allow mutators to blacken the objects they own by executing $x.b := true$. The two versions combine this action either with *delete* or *mutate*.

Version A. It may seem counter-intuitive but the simplest remedy for the above examples is to blacken the deleted object, and thus replace *delete* by:

$$\begin{array}{l} m.delete(x : Lref) : \\ \quad \langle remove\ x\ from\ m.Lref^{-}\ and\ possibly\ from\ m.Lref; \\ \quad\quad x.b := true \rangle. \end{array}$$

This action can be regarded as atomic because the only access to a shared variable is the assignment $x.b := true$.

Version B. Because deletions from *Lref* are rather implicit, we also provide a version where they are ignored. As indicated by Example 4, when implicit deletions are not observable, we need to postulate that *getLref* is atomic. Example 3 then indicates that we need to make `mutate` actions observable. Version B is much more conventional than version A. We present it here mainly for comparison with the more unusual version A.

We modify *mutate* in the following way. Mutator m gets an additional private variable $m.t$ (for target) of type *Obj*, which is initially $\perp$. The three mutator actions *delete*, *load*, *mutate* get the additional precondition $m.t = \perp$. Action *mutate* becomes:

$$\begin{array}{l} m.mutate(x, y : m.Lref, f : x.F) : \\ \quad \langle \textbf{pre}\ m.t = \perp;\ m.t := y;\ x.f := y \rangle. \end{array}$$

To re-establish the precondition $m.t = \perp$, we extend *mutate* with a second atomic action that resets $m.t := \perp$:

$$\begin{array}{l} m.blackenT() : \\ \quad \langle \textbf{if}\ m.t \neq \perp\ \textbf{then}\ m.t.b := true;\ m.t := \perp\ \textbf{end} \rangle. \end{array}$$

In other words, after *mutate*, the mutator gets the additional obligation to blacken the new target. Before this blackening, it is not able to call any of its other primitives. It follows that mutator m has the invariant $m.t = \perp \vee m.t \in m.Lref$.

2.4 When *Black* remains constant during *mark*

In the previous section we have indicated that mutators must blacken the objects they are working with according to versions A or B. Here we indicate that this is indeed enough.

Firstly, for either version, Theorem 1 remains valid. More importantly, however, either adaptation of the mutators makes the following reverse implication valid.

Theorem 2 *Assume that during a call of mark the set Black remains constant. Then this call has the postconditions Lq0 and Lq1.*

Proof We give an informal proof for version A. In Sect. 3, we give formal proofs for both versions.

In the following, we write "during loop E" to refer to all states from the one where loop E starts, up to and including the one where loop M starts. By Theorem 1, it suffices to prove that *Lq0* and *Lq1* hold in the postcondition of loop E.

We first prove that $m.Lref \subseteq Black$ holds invariantly during loop E for every mutator m. Indeed, let $x \in m.Lref$ at some point during loop E. If x is deleted from $m.Lref$ during the remainder of *mark*, it is painted black by *delete*. Otherwise, it is painted black by loop M. Because *Black* does not change during *mark*, it follows that $x.b$ holds at the point of consideration. Since $Common \subseteq Black$, this proves that *Lq0* holds invariantly during loop E, and in particular at its postcondition.

Now consider an edge (x, f) such that $x.b \wedge \neg x.f.b$ holds in the postcondition of loop E. Let $y = x.f$ at this point. Because *Black* does not change, we have $x.b \wedge \neg y.b$ continuously during loop E. If $x.f = y$ would also hold continuously during loop E, then loop E would have blackened y. Therefore, $x.f := y$ has been executed during loop E by some mutator m in *mutate*, and hence with $y \in m.Lref \subseteq Black$, which contradicts $\neg y.b$. This proves that the postcondition of loop E satisfies *Lq1*. □

Remark When the order of loops E and M in *mark* is reversed, Theorem 2 is no longer valid.

For version A, this is shown by the following scenario. Let there be three objects x, y, z with initially $Black = \{x, y\}$ and $x.f = z$. Let mutator m initially have $m.Lref = \{x, y\}$. Loop M preserves this situation. Then m performs $load(x, f)$ followed by $mutate(x, y, f)$, so that $m.Lref = \{x, y, z\}$ and $x.f = y$. Then loop E is executed without blackening any white node. In this scenario the adapted variation of *mark* does not blacken any white nodes, while in the postcondition $m.Lref$ contains the white node z.

The same scenario works for version B. This is the reason why the algorithm of [11] needs more than one nonblackening call of *mark*.

2.5 The complete marking phase

During the mutator actions and execution of *mark*, the set *Black* cannot shrink. In order to test that it remains constant, it suffices therefore to count its number of elements. For this purpose, it is easy to implement some procedure *blackCount* with the specification

$$\begin{array}{ll} (3) & blackCount() \textbf{ returns } n : \mathbb{N} \\ & \{\#Black^- \leq n \leq \#Black^+\} \end{array}$$

where $\#Black^-$ and $\#Black^+$ are the cardinalities of the sets *Black* in the precondition and the postcondition of *blackCount*, respectively. Note that procedure *blackCount* is supposed to execute concurrently with the mutator actions.

Using variables *obc* and *bc* for (old) black count, the marking phase can now be implemented as

```
markingPhase() =
    markApp(); bc := blackCount();
    do
        mark(); obc := bc; bc := blackCount()
    while obc ≠ bc.
```

It has the loop invariants

$$Common \subseteq Black \subseteq Reach^0,$$
$$bc \leq \#Black.$$

Whenever the marking phase terminates, it has the postcondition $Reach \subseteq Black$ because of Theorem 2. Since *bc* never decreases and is bounded by the total number of objects, the loop body will eventually keep *bc* constant. Then the loop terminates.

Remark For the sake of efficiency, one may replace the call of *getLref* in the body of loop M in *mark* by a procedure that only blackens all elements of $m.Lref$ without delivering the set *obs*. This is also correct, for either version, because mutator m is allowed to blacken all objects in *obs* and, if it has done so, the blackening **for** loop over *obs* is superfluous and can be removed.

2.6 Allocation, the free list, and the *sweep*

Even if there is only one mutator, allocation requires waiting when the free list is empty. When several concurrent mutators are present, the mutators must access the free list under mutual exclusion or some other form of synchronization. We here present a proposal in which the fields of `free` can be accessed by atomic actions that are slightly bigger than atomic reading and atomic writing.

Recall that $\texttt{free}.F \neq \emptyset$ and $\bot.F = \emptyset$. In order to construct lists, we assume that all other objects x have a field $\mathit{next} \in x.F$. There are as many free lists as `free` has fields. By giving `free` several children, one may hope to eliminate or alleviate congestion, possibly by also applying randomization and guided distribution of the white nodes. Alternatively, one may prefer a single free list, i.e., to give `free` a single child. For the moment, we can keep the options open.

Every mutator has a private variable *al* to hold the result of allocation. By convention, the value of $m.\mathit{al}$ is always an element of $m.\mathit{Lref}$. The fields of `free` can be accessed by the atomic command:

$$
\begin{array}{l}
m.\mathit{get}() : \\
\quad \langle \mathit{choose}\, f \in \texttt{free}.F \ \mathit{with}\ \texttt{free}.f \neq \bot; \\
\quad \mathit{al} := \texttt{free}.f;\ \texttt{free}.f := \bot \rangle.
\end{array}
$$

Mutators can fill empty slots of the free lists by the atomic command:

$$
\begin{array}{l}
\mathit{store}(x : m.\mathit{Lref}) : \\
\quad \langle \mathit{choose}\, f \in \texttt{free}.F \ \mathit{with}\ \texttt{free}.f = \bot;\ \texttt{free}.f := x \rangle.
\end{array}
$$

When a process that needs to *store* does not find an empty slot, it can execute *get* on a full slot, combine the two lists to be stored and try to *store* the combination.

A mutator m can allocate a single free object by the sequence of two atomic actions:

$$m.\mathit{get}();\ \mathit{store}(m.\mathit{al}.\mathit{next}).$$

If it needs more than a single object and the list attached to *al* is long enough, the second statement can be replaced by $\mathit{store}(z)$ where z is a deeper descendant of $m.\mathit{al}$. This also allows the well-known approach to let the mutators keep medium-sized private caches from which to allocate small objects without synchronization (as suggested by one referee).

When the collector has concluded the marking phase, it executes the *sweep*, i.e., it collects all white nodes, resets all fields of them to $\bot$, builds a list of them with first object *head*, and finally includes this list into one of the free lists by means of $\mathit{store}(\mathit{head})$.

The atomic commands *get* and *store* can be implemented by semaphores, pthread primitives, or, e.g., by compare-and-swap (`CAS`). Recall that $\texttt{CAS}(\mathtt{v}, x, y)$ is the command that atomically verifies whether $\mathtt{v} = x$ and sets $\mathtt{v} := y$ and yields *true* and otherwise it does nothing and yields *false*.

$$
\begin{array}{l}
m.\mathit{get}() : \\
\quad \textbf{repeat}\ \mathit{choose}\, f \in \texttt{free}.F;\ \mathit{al} := \texttt{free}.f; \\
\quad \textbf{until}\ \mathit{al} \neq \bot\ \wedge\ \texttt{CAS}(\texttt{free}.f, \mathit{al}, \bot). \\
\mathit{store}(x : m.\mathit{Lref}) : \\
\quad \textbf{repeat}\ \mathit{choose}\, f \in \texttt{free}.F; \\
\quad \textbf{until}\ \texttt{CAS}(\texttt{free}.f, \bot, x).
\end{array}
$$

For version A, these are the only points where synchronization is needed in the algorithm. Note that this kind of synchronization is needed anyhow when concurrent mutators access a common free list.

3 Formalization

The above proofs of Theorems 1 and 2 are sketches only. In particular, the proof of Theorem 2 for version A is a behavioural proof. Such proofs are not very reliable. We therefore turn to formal methods. The formalization presented in this section closely mimicks our PVS verification at [10].

In Sect. 3.1, we first model both algorithms to make their atomic steps explicit, and we then prove that garbage collecting does not disturb the reachable part of the heap if it preserves a certain critical invariant *Jq0*. The invariance of *Jq0* for the algorithms A and B is proved in the Sects. 3.2 and 3.3, respectively. In Sect. 3.4, we show absence of memory leaks, i.e., that every unreachable object is eventually collected.

3.1 Formal modelling and safety of garbage collection

In this subsection, we treat both versions of mutator adaptation of Sect. 2.3. We need not yet distinguish between them because all arguments are equally valid and relevant in either case.

We first reformulate the code of *mark*. All three **for all** statements in *mark* ask for a set of elements that have yet to be treated in the loop. In the following code we use the sets *edges*, *lis*, and *obs* for this purpose. We use program variables *uu* and *pp* instead of *u* and *p* (as used in Sect. 2.2), in order to emphasize that they are program variables and not logical variables, e.g., when they occur in invariants.

Loop *E* consists of the statements 20, 21, 22. Loop *M* consists of 25, 26, 27, 28. We use *pc* to indicate the location of *collector*, and 30 as the return location of procedure *mark*. In the case of version A, we model the nonatomic execution of *getLref* by separating its call in 26 from its return in 27. Command 27 chooses *obs* according to specification (0) in Sect. 2.1. In this way, an arbitrary number of mutator actions can be interleaved. In the case of version B, the atomic execution of *getLref* is modelled by the assignment $obs := pp.Lref$ followed by an immediate jump to 28.

```
mark:
E: 20:   edges := Edge;
   21:   if edges = ∅ then goto 25 else
             extract some(uu, g) from edges; goto (uu.b ? 22 : 21) end;
   22:   uu.g.b := true; goto 21;
M: 25:   lis := Mut;
   26:   if lis = ∅ then goto 30 else
             extract some pp from lis;
             if version A then call getLref(pp)
             else obs := pp.Lref ; goto 28 end;
         end;
   27:   obs := getLref(pp);
   28:   if obs = ∅ then goto 26 else
             extract some uu from obs; uu.b := true; goto 28 end.
```

The remainder of the marking phase as described in Sect. 2.5 is formalized as follows. The nonatomic call to *markApp* is split over two locations 10 and 11. In particular, command 11 establishes its postcondition (2) in Sect. 2.2. The call to *blackCount* is also split over a calling point and a return point. At the return point, a number *n* is chosen nondeterministically according to the specification (3) in Sect. 2.5. It thus becomes:

markingPhase:
10: *call markApp*;
11: *return markApp*; *call blackCount*;
12: $bc := blackCount$; **goto** 20;
30: $obc := bc$; *call blackCount*;
31: $bc := blackCount$; **goto** $(obc = bc\ ?\ 32 : 20)$.

The sweep to append the white nodes to the free list is made explicit in the locations 32 up to 35. In 34, each white node *uu* is stripped of children, and the white node *uu* is pushed onto the list of *head*. As requested by one of the referees, we treat the children of *uu* one by one in a repetition, and therefore use a local variable *flis* to hold the set of fields to be treated yet. Command 35 is a *store* action as discussed in Sect. 2.6. Notice that this *store* action is innocent and vacuous if no garbage was collected.

sweep:
32: $obs := Obj$; $head := \bot$;
33: **if** $obs = \emptyset$ **then goto** 35 **else**
extract some uu from obs;
$flis := uu.F$; **goto** $(uu.b\ ?\ 33 : 34)$ **end**;
34: **if** $flis \neq \emptyset$ **then**
extract some f from flis; $uu.f := \bot$; **goto** 34
else
$uu.next := head$; $head := uu$; **goto** 33;
end;
35: *choose* $f \in \mathtt{free}.F$ *with* $\mathtt{free}.f = \bot$;
$\mathtt{free}.f := head$; **goto** 10.

The main safety property of garbage collection is that it does not change the reachable part of the heap except for making the collected garbage objects reachable again. This is expressed in:

Theorem 3 *The collector modifies the restriction of function h to the set Reach only at* $pc = 35$. *More precisely, a collector step at* $pc \neq 35$ *does not modify* $v.f$, *i.e.* $h(v, f)$, *for any* $v \in Reach$ *and* $f \in v.F$.

By code inspection, one sees that the collector only modifies function h in 34 and 35. We thus have to prove that step 34 does not modify the restriction of h to *Reach*. Because step 34 only modifies $h(uu, f)$ for some field f, it remains to prove that $uu \notin Reach$ in 34. The collector only arrives at 34 when *uu* is not black. It is therefore natural to expect the invariant:

Ij0: $pc = 34 \quad \Rightarrow \quad uu \notin Black$.

Here we introduce the naming convention that invariants have a *j* as second letter, whereas *q* is the second letter of predicates that are no invariants but may occur in invariants.

Let us assume validity of invariant *Ij0* for the moment. Then Theorem 3 would follow if we also have $Reach \subseteq Black$ at 34.

In order to prove this, we first need to formalize Theorem 1. For that purpose, we need to introduce the following predicates:

Lq2: $pc = 27 \quad \Rightarrow \quad pp.Lref^{+} \subseteq Black$,
Lq3: $pc = 28 \quad \Rightarrow \quad obs \subseteq Black$.

As explained below specification (0), the conceptual field $pp.Lref^+$ stands for the union of the values of $pp.Lref$ encountered since the call of *getLref*. We define Lq to be the conjunction of $Lq0$, $Lq1$, $Lq2$, and $Lq3$. In the case of version B, one can omit $Lq2$ because $pc \neq 27$ always holds. Theorem 1 is now formalized and extended to:

Theorem 4 *Assume that Lq holds at some moment of the collection cycle. Then Black remains constant and Lq remains valid, until the execution of* 35, 10 *and* 11.

The mechanical proof of Theorem 4 requires the following invariants:

$$Ij1: \quad pc = 22 \quad \Rightarrow \quad uu \in Black,$$
$$Ij2: \quad m.t = \bot \quad \vee \quad m.t \in m.Lref \quad \text{for all } m \in Mut.$$

Given these invariants, the proof of Theorem 4 is straightforward but cumbersome. The validity of invariant $Ij1$ is easy to verify. Invariant $Ij2$ was announced above in Sect. 2.3.

The validity of $Ij0$ is the central problem of garbage collection: at 34, the white node uu must not be reachable for the mutators. As announced in Sect. 2.2, predicate Lq implies $Access \subseteq Black$. Therefore preservation of $Ij0$ under the mutator actions follows from

$$Ij3: \quad 32 \leq pc \quad \Rightarrow \quad Lq.$$

In 31, the *collector* goes to 32 if and only if *obc* equals the result of *blackCount*. Therefore, preservation of $Ij3$ at 31 follows from the invariant

$$Ij4: \quad pc = 31 \quad \Rightarrow \quad Lq \ \vee \ obc + 1 \leq bcUnder,$$

where *bcUnder* is the value of #*Black* at the call of *blackCount* in 30.

Preservation of $Ij4$ at 30 follows from

$$Jj0: \quad pc = 30 \quad \Rightarrow \quad Lq \ \vee \ bc + 1 \leq \#Black.$$

The second alternative of the consequent of $Jj0$ indicates that the set *Black* has grown since (or during) the latest counting because of the easy invariant:

$$Jj1: \quad 20 \leq pc \quad \Rightarrow \quad bc \leq \#Black.$$

The proof of invariance of $Jj0$ is delicate and differs for the two versions considered. We therefore postpone these proofs to the Sects. 3.2 and 3.3.

We can now easily prove Theorem 3. Indeed, the collector modifies function h only at locations 34 and 35. It modifies h in 34 only at object uu. It suffices therefore to show that uu is not reachable in 34. This follows from the invariants $Ij0$ and $Ij3$ and the observation that Lq implies $Reach \subseteq Black$. This concludes the proof of Theorem 3 except for the invariant $Jj0$.

Before proceeding to the proof of $Jj0$ in the versions A and B, we first note three easy invariants of *collector*:

$$Ij5: \quad 12 \leq pc \quad \Rightarrow \quad Common \subseteq Black,$$
$$Ij6: \quad 25 \leq pc \quad \Rightarrow \quad edges = \emptyset,$$
$$Ij7: \quad 30 \leq pc \quad \Rightarrow \quad lis = \emptyset.$$

3.2 Correctness for version A

We now restrict the attention to version A. It came as a surprise to us that the behavioural proof of Theorem 2 is formalized by means of invariants. We need relatively few invariants, but they are complicated. For proper management of them, we give names to several of their ingredients.

Theorem 5 *Jj0 is an invariant for version A.*

Proof We introduce the abbreviation *Pq0* for the second disjunct of the consequent of *Jj0* that expresses that *Black* has more node than counted before:

$$Pq0: \quad bc + 1 \leq \#Black.$$

To approximate *Lq1* (see 2.2), we introduce:

$$Pq1: \quad \forall (x, f) \in Edge : (x, f) \in edges \ \vee \ \neg x.b \ \vee \ x.f.b \\ \vee \ (pc = 22 \wedge (x, f) = (uu, g)).$$

Here the pairs (x, f) range over the set *Edge*. *Pq1* implies *Lq1* at 25 because *Ij6* implies that *edges* is empty. To deal with *Lq0*, we consider the following predicates that express that loop M still has to blacken some nodes:

$$Pq2: \quad \exists\, m \in lis : m.Lref \not\subseteq Black,$$
$$Pq3: \quad pc = 27 \ \wedge \ pp.Lref^{-} \not\subseteq Black,$$
$$Pq4: \quad pc = 28 \ \wedge \ obs \not\subseteq Black.$$

In *Pq3*, the conceptual field $pp.Lref^{-}$ stands for the intersection of the values of *pp.Lref* encountered up to that point.

We now introduce the invariants:

$$Aj0: \quad pc \in \{21 \ldots 25\} \ \Rightarrow \ Pq0 \ \vee \ Pq1 \ \vee \ \neg Lq0,$$
$$Aj1: \quad pc \in \{26 \ldots 30\} \ \Rightarrow \ Lq \ \vee \ Pq0 \ \vee \ Pq2 \ \vee \ Pq3 \ \vee \ Pq4.$$

Predicate *Pq2* is false at 30 because of *Ij7*. Predicates *Pq3* and *Pq4* are also false at 30. Therefore, *Jj0* follows from *Aj1* and *Ij7*. It takes more to prove *Aj0* and *Aj1*.

Predicate *Aj0* is preserved by 20, because action 20 establishes *Pq1*. Predicate *Pq1* is preserved by 21 and 22. When *Pq1* is falsified by blackening a white node, #*Black* is incremented so that *Pq0* is established because of *Jj1*. When *Pq1* is falsified by redirecting an edge to a white node y in *m.mutate*, we have $\neg Lq0$ because of $y \in m.Lref$. Predicate *Pq0* is obviously stable (i.e. when valid, it stays valid). When $\neg Lq0$ is falsified by deleting a node from *Lref*, this node is blackened because of version A of *delete*, and therefore *Pq0* is established. This proves the invariance of *Aj0*.

We turn to the proof of invariant *Aj1*. When *collector* enters 26, it comes from 25 and satisfies $Pq0 \vee Pq1 \vee \neg Lq0$ because of *Aj0*. In case of $Pq1 \wedge Lq0$, the step establishes *Lq* because of *Ij5* and *Ij6*. Predicate *Lq* is stable in $\{26 \ldots 30\}$ because of Theorem 4. Predicate *Pq0* is also stable in $\{26 \ldots 30\}$. In the case of $\neg Lq0$, predicate *Pq2* is established. When *Pq2*, *Pq3*, *Pq4* are falsified by blackening a white node, *Pq0* is established. When *Pq2* or *Pq3* are falsified by deleting a white node from *Lref*, this node is blackened because of version A of *delete*. When *Pq2* is falsified by extracting a mutator from *lis*, *Pq3* is established. When *Pq3* is falsified by executing 27, *Pq4* is established. When *Pq4* is falsified by extracting a node from *lis*, this node is blackened. This proves the invariance of *Aj1*, and thus concludes the proof of *Jj0* for version A. □

3.3 Correctness for version B

The key to version B is that a mutator m with $m.t \neq \bot$ must execute *blackenT*$(m.t)$ before it may delete objects from *m.Lref*. It follows that the value of $m.t$ plays an important role in the invariants for this version.

Theorem 6 *Jj0 is an invariant for version B.*

Proof Recall that $Jj0 \equiv (pc = 30 \Rightarrow Lq \vee Pq0)$, where $Pq0 : bc + 1 \leq \#Black$ expresses that *Black* has more nodes than counted before. We weaken *Pq0* to *Qq0* given by

$$\begin{array}{ll} Qq0: & Pq0 \ \vee \ Qq2 \ \vee \ Pq4, \\ Qq2: & \exists\, m \in Mut : m.t \neq \bot \ \wedge \ \neg m.t.b \ \wedge \ (pc \leq 25 \ \vee \ m \in lis). \end{array}$$

Here *Qq2* is an analogue of *Pq2* and it expresses that some mutator has a target that needs to be blackened and will be blackened in or before 28, and *Pq4* is the predicate used in the proof for version A that expresses that some object is to be blackened at 28.

While $20 \leq pc \leq 30$, predicate *Qq0* is stable, i.e., once valid it remains valid, because of *Jj1* and *Ij2*. Indeed, object $m.t$ must be blackened before it can be reset to $\bot$, and blackening establishes *Pq0* because of *Jj1*. On the other hand, if *Qq2* holds and m is removed from *lis*, then *Pq4* is established because of *Ij2*. White objects are only removed from *obs* when they are also blackened. Again this establishes *Pq0*.

We now claim the invariants:

$$\begin{array}{lll} Bj0: & 21 \leq pc \leq 30 & \Rightarrow \quad Pq1 \ \vee \ Qq0, \\ Bj1: & 26 \leq pc \leq 30 & \Rightarrow \quad Qq0 \ \vee \ (\forall\, m : m \in lis \ \vee \ m.Lref \subseteq Black). \end{array}$$

The importance of these invariants is that *Bj0*, *Bj1*, and *Ij5*, *Ij6*, and *Ij7* together imply *Jj0*. Indeed, the conjunction $Qq0 \ \wedge \ pc = 30$ implies *Pq0* because of *Ij7*. Also using *Ij6*, we get that *Bj0* implies $pc = 30 \ \Rightarrow \ Lq1 \vee Pq0$. Similarly, using *Ij5*, we get that *Bj1* implies $pc = 30 \ \Rightarrow \ Lq0 \vee Pq0$. The predicates *Lq2* and *Lq3* hold trivially at 30.

It remains to show that *Bj0* and *Bj1* are invariants. For this purpose, we first show that every step preserves *Bj0* under assumption of *Bj1* in the precondition. When *collector* enters 21, it establishes *Pq1*, so that *Bj0* is preserved. Now suppose $21 \leq pc \leq 30$. Since *Qq0* is stable, we may assume the precondition $Pq1 \ \wedge \ \neg Qq0$. When *Pq1* is falsified by blackening a white node, *Pq0* is established.

When *Pq1* is falsified by $m.mutate(x, y, f)$, we have $\neg y.b$ and the step establishes $m.t = y$. We have $y \neq \bot$ because of *Ij5*. Predicate *Qq2* is established when $pc \leq 25$ or $m \in lis$. Otherwise, by *Bj1* and $y \in m.Lref$, we have *Qq0*. This implies that *Bj0* is preserved.

We next prove that every step preserves *Bj1* under assumption of *Bj0* and *Bj1* in the precondition. When *pc* becomes 26, the collector establishes $m \in lis$. Now assume $26 \leq pc \leq 30$. Since *Qq0* is stable, we may assume *Pq1* and $\neg Qq0$ and $m \in lis$ or $m.Lref \subseteq Black$ in the precondition. The condition $m \in lis$ is falsified by 26 when $pp = m$ is chosen. Then, either $m.Lref \subseteq Black$ holds or condition *Pq4* is established. If $m \notin lis$, the condition $m.Lref \subseteq Black$ is threatened only when mutator m executes $m.load(x, f)$ to obtain a new white object. We then have $x \in m.Lref$ and $x.b \wedge \neg x.f.b$. Then *Pq1* and *Ij6* lead to a contradiction. This proves that *Bj1* is preserved. □

3.4 Progress: absence of memory leaks

Every object unreachable at the start (at 10) of a collector cycle is collected at the end of this cycle. This is expressed by specifying the following postcondition of action 35:

$$\textbf{post}(35): \quad Obj = Reach \cup Reach^0.$$

Here, $Reach^0$ is the value of set *Reach* at the latest execution of 10. Note that we cannot express this as an invariant, since the equality is not stable: when the collector remains at 10, the set *Reach* can shrink because of mutator actions.

Theorem 7 *Step* 35 *establishes* **post**(35) : $Obj = Reach \cup Reach^0$.

Proof Every object $x \notin Reach^0$ remains white during the marking phase because no nodes outside $Reach^0$ are blackened, as expressed in the invariant

$$Mj0: \quad 12 \leq pc \quad \Rightarrow \quad Black \subseteq Reach^0,$$

In order to prove *Mj0* we need:

$$Mj1: \quad 11 \leq pc \quad \Rightarrow \quad Reach \subseteq Reach^0,$$
$$Mj2: \quad 11 \leq pc < 33 \ \wedge \ x \in Reach^0 \quad \Rightarrow \quad x.f \in Reach^0.$$

In particular, we need *Mj1* because mutators can blacken reachable nodes, and *Mj2* because loop E of the collector transfers blackness along edges. Preservation of *Mj0* at 28 follows from:

$$Mj3: \quad pc = 28 \quad \Rightarrow \quad obs \subseteq Reach^0,$$
$$Mj4: \quad pc = 27 \quad \Rightarrow \quad pp.Lref^+ \subseteq Reach^0.$$

In the proof of invariance of *Mj3* we need *Mj4* because, in 28, the set *obs* is chosen as a subset of $pp.Lref^+$ (note that we cannot postulate $pp.Lref^+ \subseteq Reach$).

In view of *Mj0*, it remains to show that, at 35, all white objects become reachable because they enter the free list. At 35, the list starting in *head* is attached to `free`. This list is being built in commands 33 and 34. To describe it, we define the predicate *HL* (for head list) to express that a set U of objects is the contents of this list, and that the objects in this list have $\perp$ at all fields $f \neq next$:

$$\begin{aligned} HL(U) \equiv \exists\, n \in \mathbb{N}, g \in ([0 \ldots n] \to Obj) : \\ U = \{g(i) \mid 0 \leq i < n\} \ \wedge \ g(0) = head \ \wedge \ g(n) = \perp \\ \wedge\, \forall\, i : 0 \leq i < n \ \Rightarrow \ (g(i+1) = g(i).next \\ \wedge \ \forall\, f \in g(i).F : f \neq next \ \Rightarrow \ g(i).f = \perp). \end{aligned}$$

Because `free` is reachable, it remains to prove that, at 35, the list of *head* consists of all white objects, as expressed in the invariant

$$Nj0: \quad pc = 35 \quad \Rightarrow \quad HL(\{x \mid \neg x.b\}).$$

As *pc* only becomes 35 in 33 when *obs* is empty, we need the invariant:

$$Nj1: \quad pc = 33 \quad \Rightarrow \quad HL(\{x \mid \neg x.b \ \wedge \ x \notin obs\}),$$

which is obviously established in the form $HL(\emptyset)$ by 32. In order to see that *Nj1* is preserved when the loop at 34 terminates, we postulate the loop invariants:

$$Nj2: \quad pc = 34 \quad \Rightarrow \quad HL(\{x \mid x \neq uu \ \wedge \ \neg x.b \ \wedge \ x \notin obs\}),$$
$$Nj3: \quad pc = 34 \quad \Rightarrow \quad uu \notin obs,$$
$$Nj4: \quad pc = 34 \quad \Rightarrow \quad \forall\, f \in uu.F : f \in flis \ \vee \ uu.f = \perp.$$

Preservation of *Nj0* follows from *Nj1*. Preservation of *Nj1* follows from *Ij0*, *Nj2*, and *Nj3*. Preservation of *Nj2*, *Nj3*, and *Nj4* is relatively easy. This concludes the proof. □

4 Using the proof assistant PVS

The formalization in Sect. 3 was performed concurrently with the development of our verification [10] with the proof assistant PVS [17]. The verification consists of four theories.

The main theory `conGarCol` uses three nonempty uninterpreted types for *Obj*, *Field*, and *Mut*, and a relation for *Edge*:

```
Object, Field, Mut: TYPE+
isEdge: pred[[Object, Field]]
```

It then declares the state space as a set of tuples. States have many components, more than indicated here:

```
state: TYPE = [#
  h: [(isEdge) -> Object],
  priv, privUnder, privUpper: [Mut -> pred[Object]],
  target: [Mut -> Object],
  ...
  black: pred[Object],
  pc: nat,
  pp: Mut,
  ...
#]
```

We use `priv(m)` for $m.Lref$ and `privUnder(m)` for $m.Lref^-$ and `target(m)` for $m.t$.

Every atomic step of the system is represented by a relation between states. In the PVS specification language, we define a relation `mutator` which includes the algorithm steps given in Sect. 2.1 as:

```
x, y: VAR state
m: VAR Mut
u: VAR Object
bit: VAR bool

mutator(m)(x, y): bool =
  (EXISTS u, bit: delete(m, u, bit)(x, y))
  OR (EXISTS u, f: inspect(m, u, f)(x, y))
  OR (EXISTS u, v, f: mutate(m, u, v, f)(x, y))
  OR blackenTarget(m)(x, y)
```

The first alternative here is the mutator step *m.delete* of Sect. 2.1. It is represented by the definition:

```
delete(m, u, bit)(x, y): bool =
  x`target(m) = null AND x`priv(m)(u) AND
  y = x WITH [
        `priv(m) := IF bit THEN remove(u, x`priv(m))
                    ELSE x`priv(m) ENDIF,
        `privUnder(m) := remove(u, x`privUnder(m)),
        `black := IF versA THEN add(u, x`black)
                  ELSE x`black ENDIF
      ]
```

Here, `versA` is a boolean to indicate whether we are treating version A or version B. We omit the other mutator steps here for the sake of brevity.

Similarly, we define a relation `collector` for the steps presented in Sect. 3.1 as a union of relations for the various *pc* values. For example, the collector step at $pc = 27$ is represented in PVS by:

```
step27(x, y): bool =
  x`pc = 27 AND
  EXISTS (us: pred[Object]):
    subset?(x`privUnder(x`pp), us)
    AND subset?(us, x`privUpper(x`pp))
    AND y = x WITH [
              `pc := 28,
              `npriv := us
            ]
```

The total step relation is the union of the various mutator relations and the sequence of collector relations.

After having encoded the data structure and its concurrent modifications according to the algorithm in this way, the real work begins. This consists in formulating the proof obligations and proving them. The proof obligations get the form of lemmas and theorems in the same theory as the declarations. Proving the lemmas and theorems is done interactively, and is recorded by PVS in a separate proof file.

For example, part of Theorem 3 of Sect. 3.1 is covered by

```
theorem3: THEOREM
  collector(x, y) AND iq0(x) AND iq3(x)
  AND reachable(x)(v) AND isEdge(v, f)
  IMPLIES y`h(v, f) = x`h(v, f) OR step35(x, y)
```

The main part of Theorem 3 however is that the predicates `iq0` and `iq3` (which represent *Ij0* and *Ij3*) are indeed invariants. This follows from the fact that they are included in the global invariant. In the case of version A, this is `globinvA` and its invariance is expressed in theory `conGarColA` by the two theorems

```
globinvA_init: THEOREM
  init(x) AND versA IMPLIES globinvA(x)
```

```
globinvA_step: THEOREM
  globinvA(x) AND stepA(x, y) IMPLIES globinvA(y)
```

Everything claimed in Sect. 3 has been proved in this way. For details we refer to [10].

5 Conclusions

There is still room for the design from scratch of simple but subtle concurrent garbage collection algorithms. Practical experience will be needed to validate the ideas worked out here. One will then have to reckon with the atomicity properties of the available concurrency platform. It may, e.g., be quite a challenge to implement the algorithms in the Java memory model, see [2, 12]. More generally, as suggested by one of the referees, it seems likely that our algorithms would need (e.g.) memory fences on platforms without sequential consistency.

In our experience, a proof assistant like PVS is indispensable for the design of algorithms with fine-grain concurrency. It requires much work to guide the proof assistant to prove all results required. The advantages are that one always knows precisely what has been proved and which proof obligations are still pending, and that one can try to reuse old proofs after changes of the algorithm or its encoding.

Indeed, during the design of the two algorithms, the use of the proof assistant PVS [17] was essential for preservation of consistency and confidence. The proofs of the two versions could be fruitfully combined. In fact, the whole proof took 154 lemmas, of which 27 for version A, 22 for version B, and 105 lemmas for both versions.

References

1. Andrews GR (1991) Concurrent programming, principles and practice. Addison-Wesley, Reading
2. Arnold K, Gosling J, Holmes D (2006) The Java programming language, 4th edn. Addison-Wesley, Reading
3. Azatchi H, Levanoni Y, Paz H, Petrank E (2003) An on-the-fly mark and sweep garbage collector based on sliding views. In: Proceedings of the 18th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications. ACM, New York, pp 269–281
4. Bacon D, Attanasio D, Lee H, Smith S (2001) Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. In: Proceedings of the SIGPLAN 2001 conference on programming languages design and implementation. ACM, New York, pp 92–103
5. Ben-Ari M (1984) Algorithms for on-the-fly garbage collection. ACM Trans Program Lang Syst 6:333–344
6. Dijkstra EW, Lamport L, Martin AJ, Scholten CS, Steffens EFM (1978) On-the-fly garbage collection: an exercise in cooperation. Commun ACM 21:966–975
7. Doligez D, Gonthier G (1994) Portable, unobstrusive garbage collection for multiprocessor systems. In: POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM, New York, pp 70–83
8. Gao H, Groote JF, Hesselink WH (2007) Lock-free parallel and concurrent garbage collection by mark&sweep. Sci Comput Program 64:341–374
9. Groote JF, Hesselink WH, Mauw S, Vermeulen R (2001) An algorithm for the asynchronous write-all problem based on process collision. Distrib Comput 14:75–81
10. Hesselink WH (2009) PVS script for "simple concurrent garbage collection". Available at www.cs.rug.nl/~wim/mechver/simple_gc/index.html
11. Jonker JE (1992) On-the-fly garbage collection for several mutators. Distrib Comput 5:187–199
12. Lea D (2000) Concurrent programming in Java. Addison-Wesley, Reading
13. Levanoni Y, Petrank E (2006) An on-the-fly reference-counting garbage collector for Java. ACM Trans Program Lang Syst 28:1–69
14. Lin C-Y, Hou TW (2007) A simple and efficient algorithm for cycle collection. ACM SIGPLAN Not 42(3):7–13
15. McCreight A, Shao Z, Lin C, Li L (2007) A general framework for certifying garbage collectors and their mutators. In: PLDI'07. ACM, New York, pp 468–479
16. Prensa Nieto L, Esparza L (2000) Verifying single and multi-mutator garbage collectors with Owicki-Gries in Isabelle/HOL. In: 25th symposium of mathematical foundations of computer science 2000. LNCS, vol 1893. Springer, Berlin, p 619
17. Owre S, Shankar N, Rushby JM, Stringer-Calvert DWJ (2001) PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference. http://pvs.csl.sri.com
18. Paz H, Bacon DF, Kolodner EK, Petrank E, Rajan VT (2007) An efficient on-the-fly cycle collection. ACM Trans Program Lang Syst 29:20
19. Steele GL (1975) Multiprocessing compactifying garbage collection. Commun ACM 18:495–508
20. van de Snepscheut JLA (1987) "Algorithms for on the fly-garbage-collection" revisited. Inf Process Lett 24:211–216
21. Vechev MT, Yahav E, Bacon DF (2006) Correctness-preserving derivation of concurrent garbage collection algorithms. In: Proceedings of the 2006 ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 341–353
22. Yuasa T (1990) Real-time garbage collection on general-purpose machines. J Syst Softw 11:181–198